

White-box attack resistant cryptography



Hiding cryptographic keys against the powerful attacker

Dušan Klínek, Petr Švenda {xklinec, svenda}@fi.muni.cz



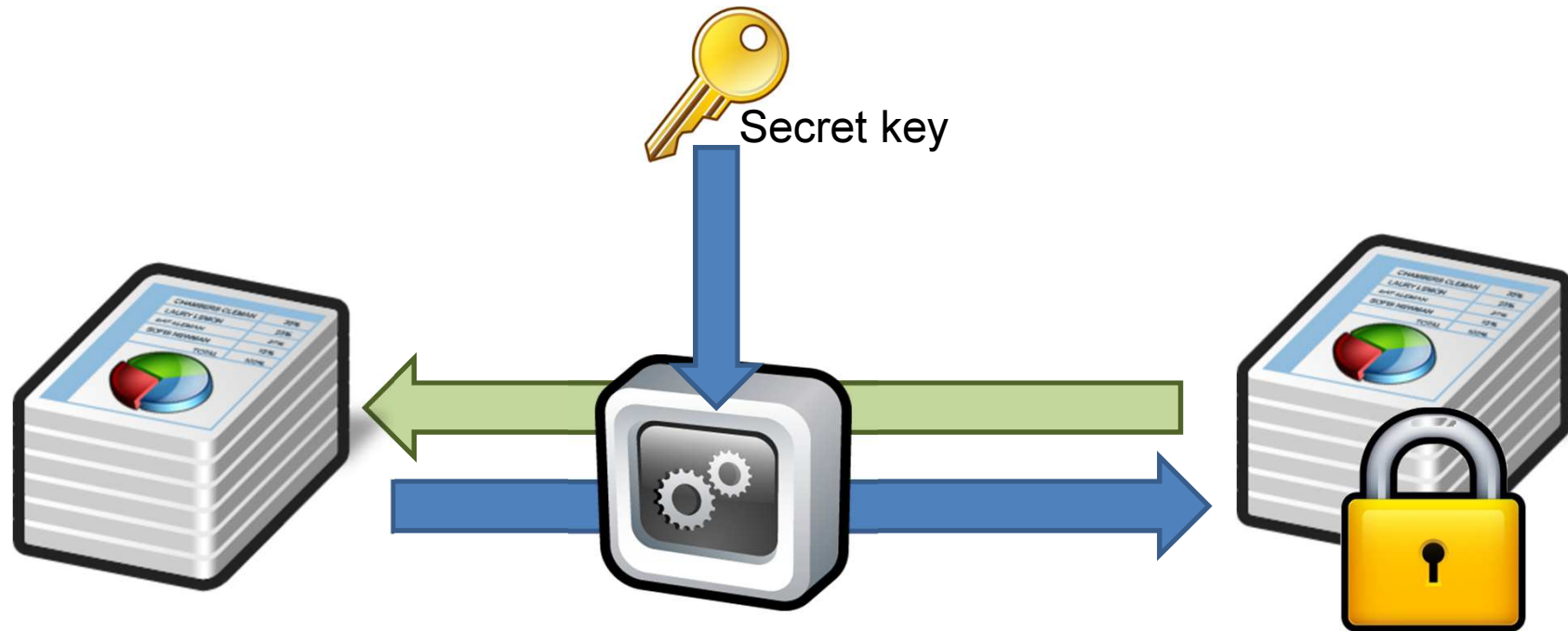
Outline

- Short intro to symmetric/asymmetric cryptography
- Classical implementations & related problems
- CEF&CED, practical problems
- Whitebox cryptography, whitebox-AES
- Available implementations & attacks
- Future work, related R&D at CROCS@FIMU

Protecting key material for cryptographic functions

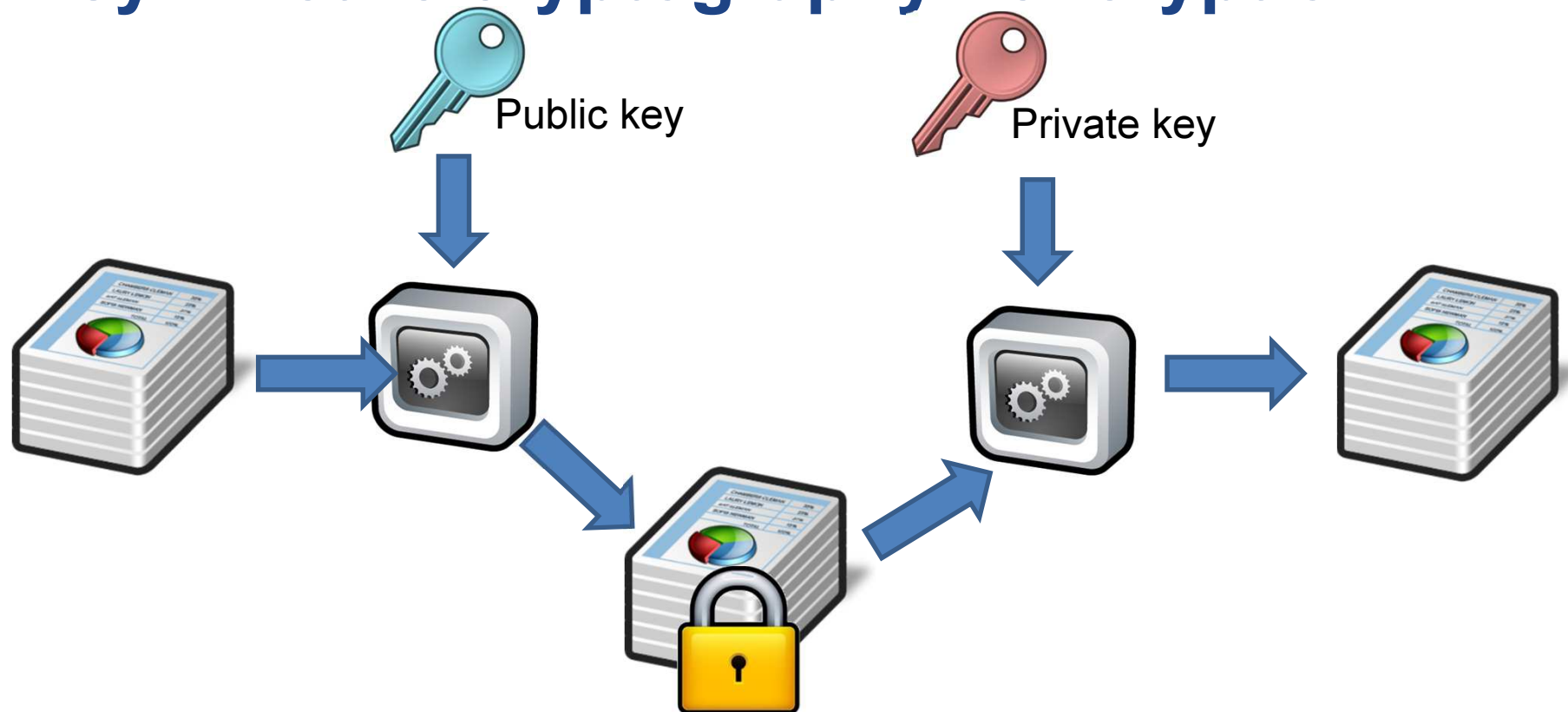
TROUBLES WITH KEYS

Cryptography with symmetric key



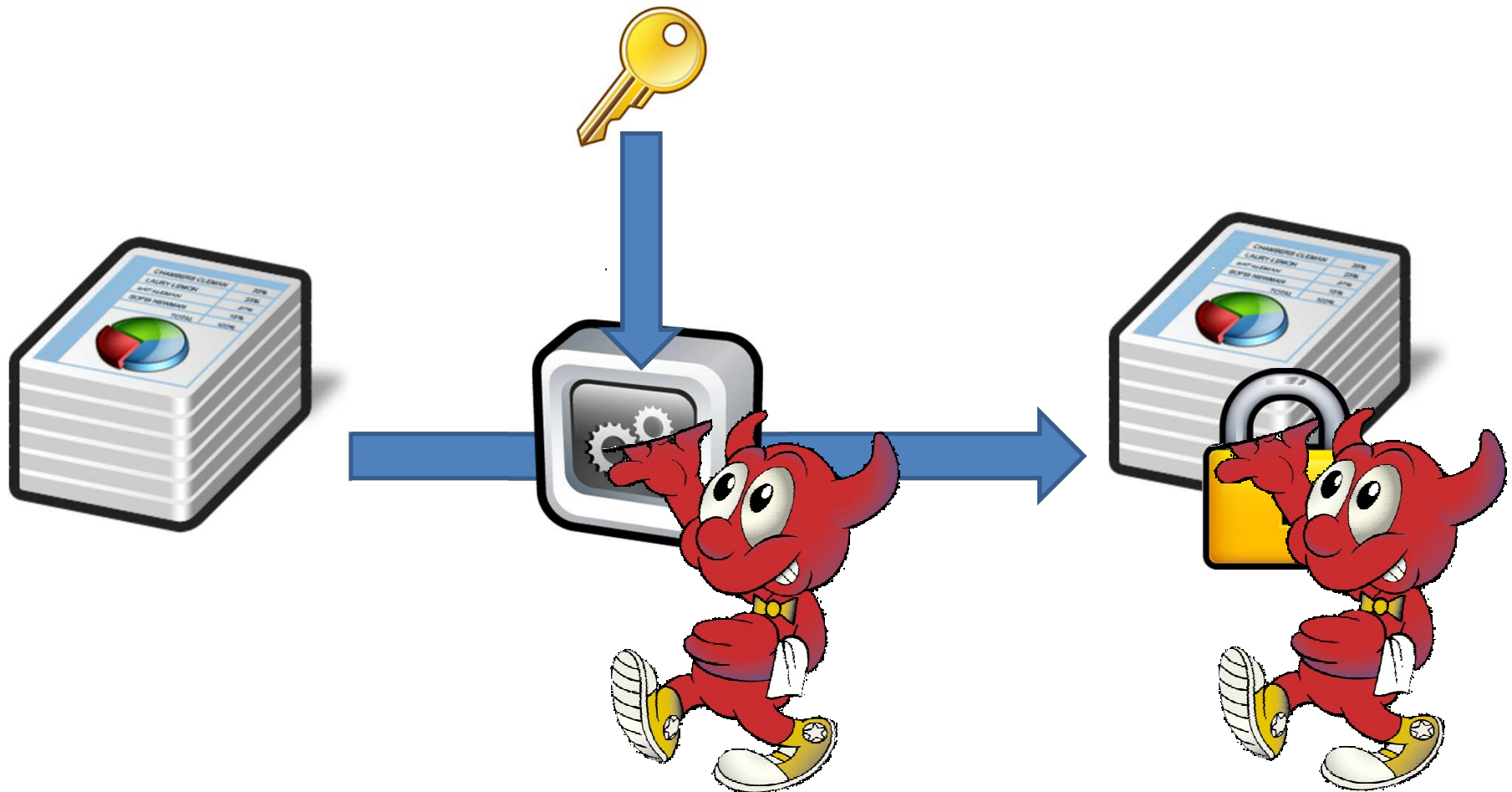
- Algorithms: DES, AES, Blowfish, IDEA, RC4...
- Key lengths: 16-32 bytes
- Usage: encryption, cryptographic checksum, auth.

Asymmetric cryptography - encryption



- Algorithms: RSA, Diffie-Hellman, ECC...
- Key lengths: 32 bytes (ECC-256) – 256 bytes (RSA-2048)
- Usage: encryption, digital signature, authentication

Standard vs. whitebox attacker model

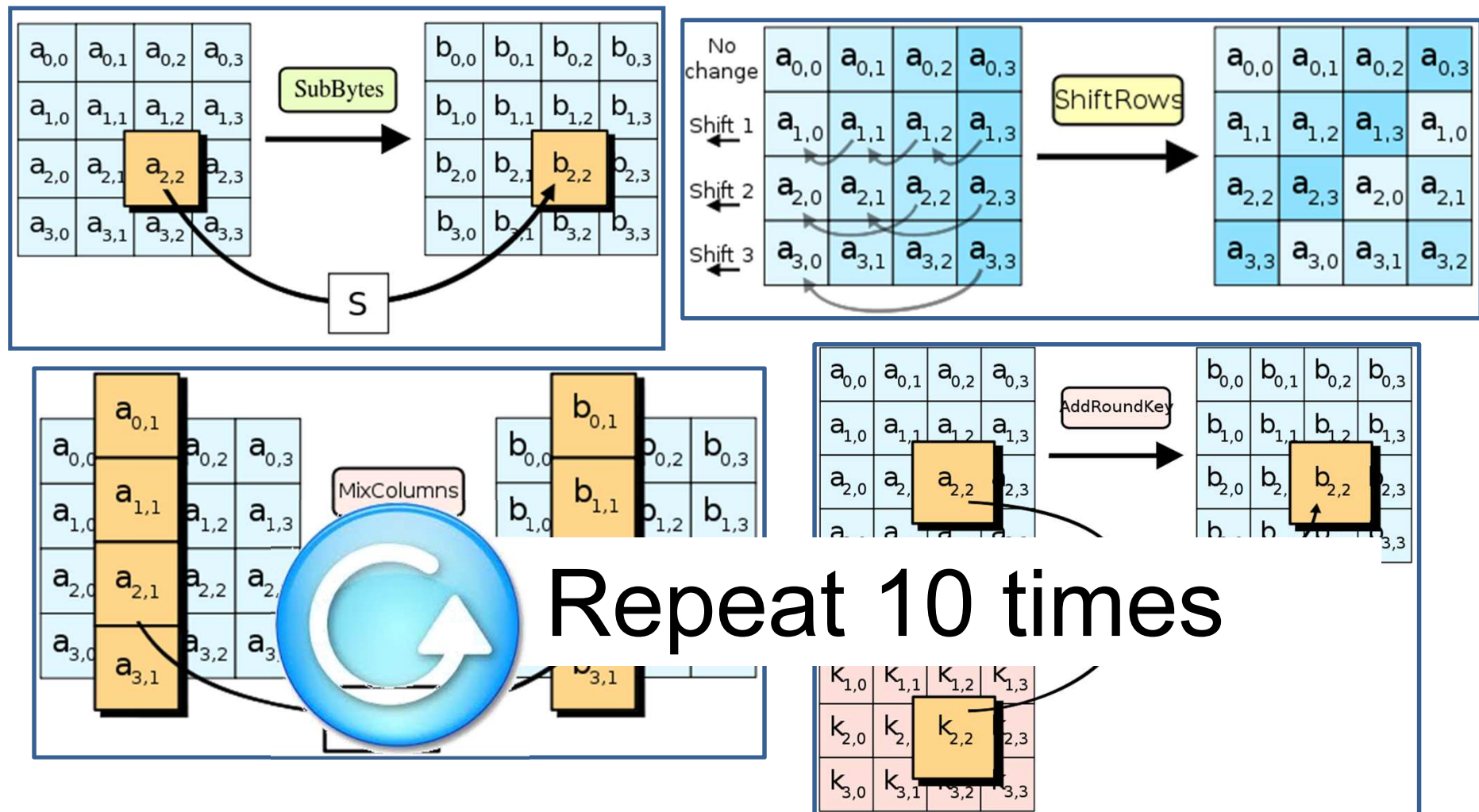


Standard AES API (PolarSSL)

```
/**
 * \brief          AES key schedule (encryption)
 *
 * \param ctx      AES context to be initialized
 * \param key      encryption key
 * \param keysize  must be 128, 192 or 256
 *
 * \return         0 if successful, or POLARSSL_ERR_AES_INVALID_KEY_LENGTH
 */
int aes_setkey_enc(aes_context *ctx, const unsigned char *key, unsigned int keysize);

/**
 * \brief          AES-ECB block encryption/decryption
 *
 * \param ctx      AES context
 * \param mode     AES_ENCRYPT or AES_DECRYPT
 * \param input    16-byte input block
 * \param output   16-byte output block
 *
 * \return         0 if successful
 */
int aes_crypt_ecb(aes_context *ctx,
                  int mode,
                  const unsigned char input[16],
                  unsigned char output[16]);
```

Advanced Encryption Algorithm



Standard AES - usage

```
void simpleAES() {
    unsigned char key[32];
    unsigned char buf[16];
    aes_context ctx;

    memset( buf, 1, sizeof(buf));
    memset( &ctx, 0, sizeof(ctx));

    // Set the key
    sprintf((char*)key, "%s", "SecurePassword:nbu123");
    aes_setkey_enc( &ctx, key, 128);

    printf("Input: ");
    for (int i = 0; i < AES_BLOCK_SIZE; i++) printf("%2x", buf[i]);
    printf("\n");

    // Encrypt one block
    aes_crypt_ecb( &ctx, AES_ENCRYPT, buf, buf );
    printf("Output: ");
    for (int i = 0; i < AES_BLOCK_SIZE; i++) printf("%x", buf[i]);
}
```

OllyDbg – key value is static string

OllyDbg - AES_PolarSSL.exe

File View Debug Plugins Options Window Help

CPU - main thread, module AES_Pola

01102EFF . 8955 E4 MOV DWORD PTR SS:[EBP-1C],EDX
 01102F02 . A1 08411D01 MOV EAX,DWORD PTR DS:[11D4108]
 01102F07 . 8945 E8 MOV DWORD PTR SS:[EBP-18],EAX
 01102F0A . 8A0D 0C411D01 MOV CL,BYTE PTR DS:[11D410C]
 01102F10 . 894D EC MOV BYTE PTR SS:[EBP-14],CL
 01102F13 . 68 10411D01 PUSH AES_Pola.011D4110
 01102F18 . 68 28411D01 PUSH AES_Pola.011D4128
 01102F1D . 8D55 D8 LEA EDI,DWORD PTR SS:[EBP-28]
 01102F20 . 52 PUSH EDI
 01102F21 . FF15 90401D01 CALL DWORD PTR DS:[<MSUCR110.sprintf>]
 01102F27 . 83C4 0C ADD ESP,0C
 01102F2A . 68 80000000 PUSH 80
 01102F2F . 8D45 D8 LEA EDI,DWORD PTR SS:[EBP-28]
 01102F33 . 50 PUSH EAX
 01102F38 . 8D8D 80FEFFFF LEA ECX,DWORD PTR SS:[EBP-180]
 01102F39 . 51 PUSH ECX
 01102F3A . E8 C1E0FFFF CALL AES_Pola.aes_setkey_end
 01102F3F . 83C4 0C ADD ESP,0C
 01102F42 . 68 2C411D01 PUSH AES_Pola.011D412C
 01102F47 . FF15 98401D01 CALL DWORD PTR DS:[<MSUCR110.printf>]
 01102F4D . 83C4 04 ADD ESP,4
 01102F50 . C745 FC 000000 MOV DWORD PTR SS:[EBP-4],0
 01102F57 . EB 09 JMP SHORT AES_Pola.011D2F62
 01102F59 . 8B55 FC MOV EDI,DWORD PTR SS:[EBP-4]
 01102F5C . 83C2 01 ADD EDI,1
 01102F5F . 8955 FC MOV DWORD PTR SS:[EBP-4],EDI
 01102F62 . 337D FC 10 CMP DWORD PTR SS:[EBP-4],10
 01102F66 . 7D 19 JGE SHORT AES_Pola.011D2F81
 01102F68 . 8B45 FC MOV EAX,DWORD PTR SS:[EBP-4]
 01102F6B . 0FB64C05 98 MOVZX ECX,BYTE PTR SS:[EBP+EAX-68]
 01102F70 . 51 PUSH ECX
 01102F71 . 68 34411D01 PUSH AES_Pola.011D4134
 01102F76 . FF15 98401D01 CALL DWORD PTR DS:[<MSUCR110.printf>]
 01102F7C . 83C4 08 ADD ESP,8
 01102F7F . EB D8 JMP SHORT AES_Pola.011D2F59
 01102F81 . 68 38411D01 PUSH OFFSET AES_Pola.load_config_used
 01102F86 . FF15 98401D01 CALL DWORD PTR DS:[<MSUCR110.printf>]
 01102F8C . 83C4 04 ADD ESP,4
 01102F8F . 8D55 98 LEA EDI,DWORD PTR SS:[EBP-68]
 01102F92 . 52 PUSH EDI
 01102F93 . 8D45 98 LEA EAX,DWORD PTR SS:[EBP-68]
 01102F96 . 50 PUSH EAX
 011D4110=AES_Pola.011D4110 (ASCII "SecurePassword:nbu123")

aes_polarssl.cpp:862. sprintf((char*)key, "%s", "SecurePassword:nbu123");

Address Hex dump ASCII
 01105000 01 00 00 00 00 00 00 00 0.....

Text strings referenced in AES_Pola:text

Address	Disassembly	Text string
011D1000	PUSH EBP	(Initial CPU selection)
011D2F13	PUSH AES_Pola.011D4110	ASCII "SecurePassword:nbu123"
011D2F18	PUSH AES_Pola.011D4128	ASCII "%s"
011D2F42	PUSH AES_Pola.011D412C	ASCII "Input: "
011D2F71	PUSH AES_Pola.011D4134	ASCII "%2x"
011D2FA8	PUSH AES_Pola.011D413C	ASCII "Output: "
011D2FD7	PUSH AES_Pola.011D4148	ASCII "%x"

0020FACC 011D326D AES_Pola.<ModuleEntryPoint>
 0020F0A0 00000000

OlllyDbg – key is visible in memory

The screenshot shows the OlllyDbg interface for the process AES_PolarSSL.exe. The CPU window displays the main thread's instructions, and a memory dump window shows the contents of memory address 0020B000..0020FFFF.

CPU - main thread, module AES_Pola

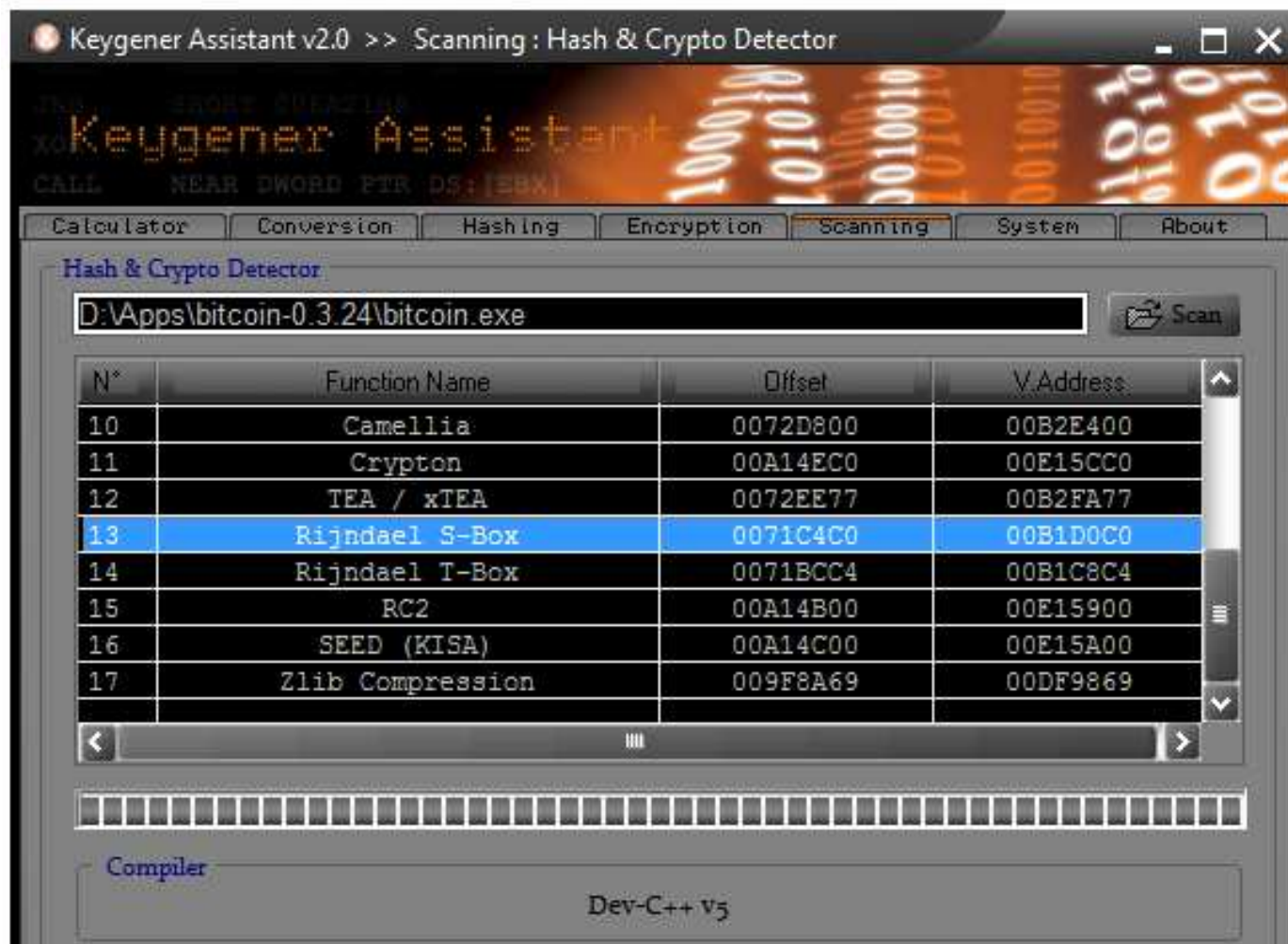
Address	Disassembly
011D1000	\$ 55 PUSH EBP
011D1001	. 8BEC MOV EBP,ESP
011D1003	. 83EC 10 SUB ESP,10
011D1006	. 56 PUSH ESI
011D1007	. 833D 20501D01 CMP DWORD PTR DS:[aes_init_done],0
011D100E	✓ 75 0F JNZ SHORT AES_Pola.011D101F
011D1010	. E8 1B1A0000 CALL AES_Pola.aes_gen_tables
011D1015	. C705 20501D01 MOV DWORD PTR DS:[aes_init_done],1
011D101F	> 8B45 10 MOV EAX,DWORD PTR SS:[EBP+10]
011D1022	. 8945 F4 MOV DWORD PTR SS:[EBP-C],EAX
011D1025	. 817D F4 800000 CMP DWORD PTR SS:[EBP-C],80
011D102C	✓ 74 14 JE SHORT AES_Pola.011D1042
011D102E	. 817D F4 C00000 CMP DWORD PTR SS:[EBP-C],0C0
011D1035	✓ 74 16 JE SHORT AES_Pola.011D104D
011D1037	. 817D F4 000100 CMP DWORD PTR SS:[EBP-C],100
011D103E	✓ 74 18 JE SHORT AES_Pola.011D1058
011D1040	✓ EB 21 JMP SHORT AES_Pola.011D1063
011D1042	> 8B4D 08 MOV ECX,DWORD PTR SS:[EBP+8]
011D1045	. C701 0A000000 MOV DWORD PTR DS:[ECX],0A
011D1048	✓ EB 20 JMP SHORT AES_Pola.011D106D
011D104D	> 8B55 08 MOV EDX,DWORD PTR SS:[EBP+8]
011D1050	. C702 0C000000 MOV DWORD PTR DS:[EDX],0C
011D1056	✓ EB 15 JMP SHORT AES_Pola.011D106D
011D1058	. 8B45 08 MOV EAX,DWORD PTR SS:[EBP+8]
011D105B	. C700 0E000000 MOV DWORD PTR DS:[EAX],0E
011D1061	✓ EB 0A JMP SHORT AES_Pola.011D106D
011D1063	> B8 00F8FFFF MOV EAX,-800
011D1068	✓ E9 0B060000 JMP AES_Pola.011D1678
011D106D	> 8B4D 08 MOV ECX,DWORD PTR SS:[EBP+8]
011D1070	. 83C1 08 ADD ECX,8
011D1073	. 894D FC MOV DWORD PTR SS:[EBP-4],ECX
011D1076	. 8B55 08 MOV EDX,DWORD PTR SS:[EBP+8]
011D1079	. 8B45 FC MOV EAX,DWORD PTR SS:[EBP-4]
011D107C	. 8942 04 MOV DWORD PTR DS:[EDX+4],EAX
011D107F	. C745 F8 000000 MOV DWORD PTR SS:[EBP-8],0
011D1086	✓ EB 09 JMP SHORT AES_Pola.011D1091
011D1088	> 8B4D F8 MOV ECX,DWORD PTR SS:[EBP-8]
011D108B	. 83C1 01 ADD ECX,1
011D108E	. 894D F8 MOV DWORD PTR SS:[EBP-8],ECX
011D1091	> 8B55 10 MOV EDX,DWORD PTR SS:[EBP+10]
011D1094	. C1FA 05 SAR EDX,5
011D1097	. 3955 F8 CMP DWORD PTR SS:[EBP-8],EDX

ESI=00000001

Dump - 0020B000..0020FFFF

Address	Hex	ASCII
0020F97F	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020F98F	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020F99F	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020F9AF	00 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	0000000000000000
0020F9BF	01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	0000000000000000
0020F9CF	01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	0000000000000000
0020F9DF	01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	0000000000000000
0020F9EF	01 53 65 63 75 72 65 50 61 73 73 77 6F 72 64 3A	0SecurePassword:
0020F9FF	6E 62 75 31 32 33 00 1D 01 58 72 1D 01 5C 72 1D	nbw123..#0Xr#0\#
0020FA0F	01 60 72 1D 01 00 00 00 00 20 FA 20 00 F8 2F 1D	0*#0.....°/#
0020FA1F	01 60 FA 20 00 05 32 1D 01 01 00 00 00 08 E1 4F	0*0*#2#00...ip0
0020FA2F	00 F8 7C 4F 00 7C 75 74 0B 00 00 00 00 00 00 00	..°!0.iut...4..%0.
0020FA3F	00 00 E0 FD 7E 00 00 00 00 34 FA 20 00 58 01 00	..02"....4..%0.
0020FA4F	00 9C FA 20 00 89 36 1D 01 34 CD 49 0A 00 00 00	.E..e6#04=I r...
0020FA5F	00 6C FA 20 00 AA 33 68 76 00 E0 FD 7E AC FA 20	.l..-3hu.02"%...
0020FA6F	00 F2 9E EE 76 00 E0 FD 7E 77 18 E1 52 00 00 00	.=x~v.02"%w#R...
0020FA7F	00 00 00 00 00 E0 FD 7E 00 00 00 00 00 00 0002".....
0020FA8F	00 00 00 00 78 FA 20 00 00 00 00 00 FF FF FFX:.....
0020FA9F	FF 05 71 F2 76 EB 27 2C 24 00 00 00 00 C4 FA 20	'q=vü',\$......-
0020FAAF	00 C5 9E EE 76 6D 32 1D 01 00 E0 FD 7E 00 00 00	.+x~vm2#0.02"....

What if AES usage is somehow hidden?



Whitebox attacker model

- The attacker is able to:
 - inspect and disassemble binary (static strings, code...)
 - observe/modify all executed instructions (OllyDbg...)
 - observe/modify used memory (OllyDbg, memory dump...)
- How to still protect value of cryptographic key?
- Who might be whitebox attacker?
 - Mathematician (for fun)
 - Security researcher / Malware analyst (for work)
 - DRM cracker (for fun&profit)
 - ...

Classical obfuscation and its limits

- Time-limited protection
- Obfuscation is mostly based on obscurity
 - add bogus jumps
 - reorder related memory blocks
 - transform code into equivalent one, but less readable
 - pack binary into randomized virtual machine
 - ...
- Barak's (im)possibility result (2001)
 - family of functions that will always leak some information
 - but practical implementation may exist for others

Computation with Encrypted Data and Encrypted Function

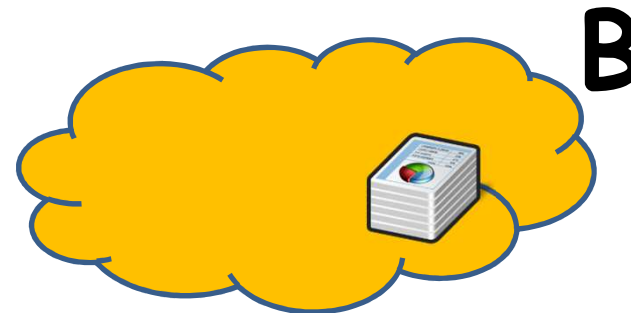
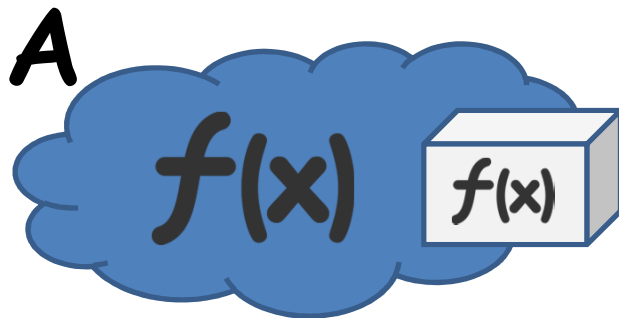
CEF&CED

Scenario

- We'd like to compute function F over data D
 - secret algorithm F or sensitive data D (or both)
- Solution with trusted environment
 - my trusted PC, trusted server, trusted cloud...
- Problem: can be cloud or client really trusted?
 - server hack, DRM, malware...
- Attacker model
 - controls execution environment (debugging)
 - sees all instructions and data executed

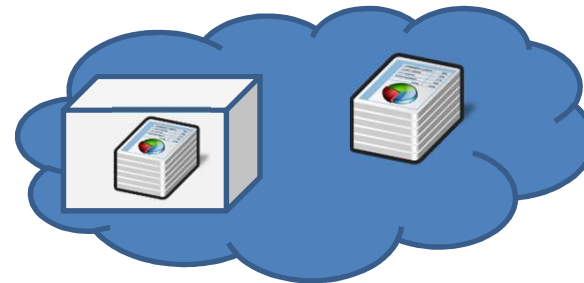
CEF

- Computation with Encrypted Function (CEF)
 - A provides function F in form of $P(F)$
 - P can be executed on B's machine with B's data D as $P(D)$
 - B will not learn function F during computation



CED

- Computation with Encrypted Data (CED)
 - B provides encrypted data D as $E(D)$ to A
 - A is able to compute its F as $F(E(D))$ to produce $E(F(D))$
 - A will not learn D

A**B**

CED via homomorphism

1. Convert your function into circuit with additions (**xor**) and multiplications (**and**) only
2. Compute addition and/or multiplication “securely”
 - an attacker can compute $E(D1+D2) = E(D1)+E(D2)$
 - but will learn neither $D1$ nor $D2$
3. Execute whole circuit over encrypted data
 - Partial homomorphic scheme
 - either addition or multiplication is possible, but not both
 - Fully homomorphic scheme
 - both addition and multiplication (unlimited)

Partial homomorphic schemes

- Example with RSA (*multiplication*)
 - $E(d_1) \cdot E(d_2) = d_1^e \cdot d_2^e \bmod m = (d_1 d_2)^e \bmod m = E(d_1 d_2)$
- Example Goldwasser-Micali (*addition*)
 - $E(d_1) \cdot E(d_2) = x^{d_1} r_1^2 \cdot x^{d_2} r_2^2 = x^{d_1+d_2} (r_1 r_2)^2 = E(d_1 \oplus d_2)$
- Limited to polynomial and rational functions
- Limited to only one type of operation (*mult* or *add*)
 - or one type and very limited number of other type
- Slow – based on modular mult or exponentiation
 - every operation equivalent to whole RSA operation

Fully homomorphic scheme (FHE)

- Holy grail - idea proposed in 1978 (Rivest et al.)
 - both addition and multiplication securely
- But no scheme until 2009 (Gentry)!
 - based on lattices over integers
 - noisy FHE usable only to few operations
 - combined with repair operation

Fully homomorphic scheme - usages

- Outsourced cloud computing and storage (FHE search)
 - Private Database Queries
 - using Somewhat Homomorphic Encryption
<http://researcher.ibm.com/researcher/files/us-shaih/privateQueries.pdf>
 - protection of the query content
- Secure voting protocols (yes/no + sum)
- Protection of proprietary info - MRI machines
 - very expensive algorithm analyzing MR data, HW protected
 - central processing restricted due to processing of private patient data
- Read more about current state of FHE
 - <http://www.americanscientist.org/issues/id.15906,y.2012,no.5,content.true,page.2,css.print/issue.aspx>

Fully homomorphic scheme - practicality

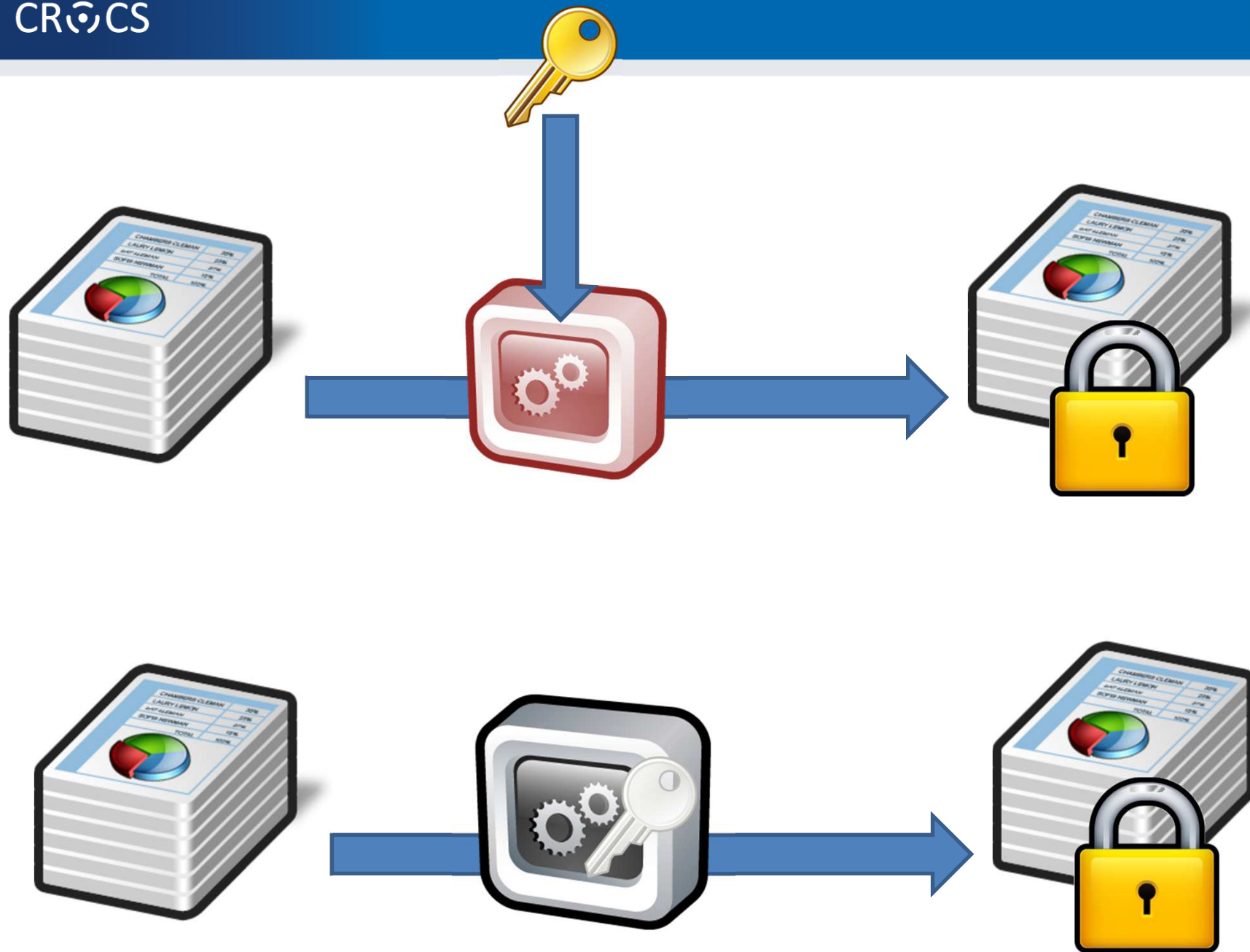
- Not very practical (yet 😊) (Gentry, 2009)
 - 2.7GB key & 2h computation for every repair operation
 - repair needed every ~10 multiplication
- FHE-AES implementation (Gentry, 2012)
 - standard PC \Rightarrow 37 minutes/block (but 256GB RAM)

Protection of cryptographic primitives

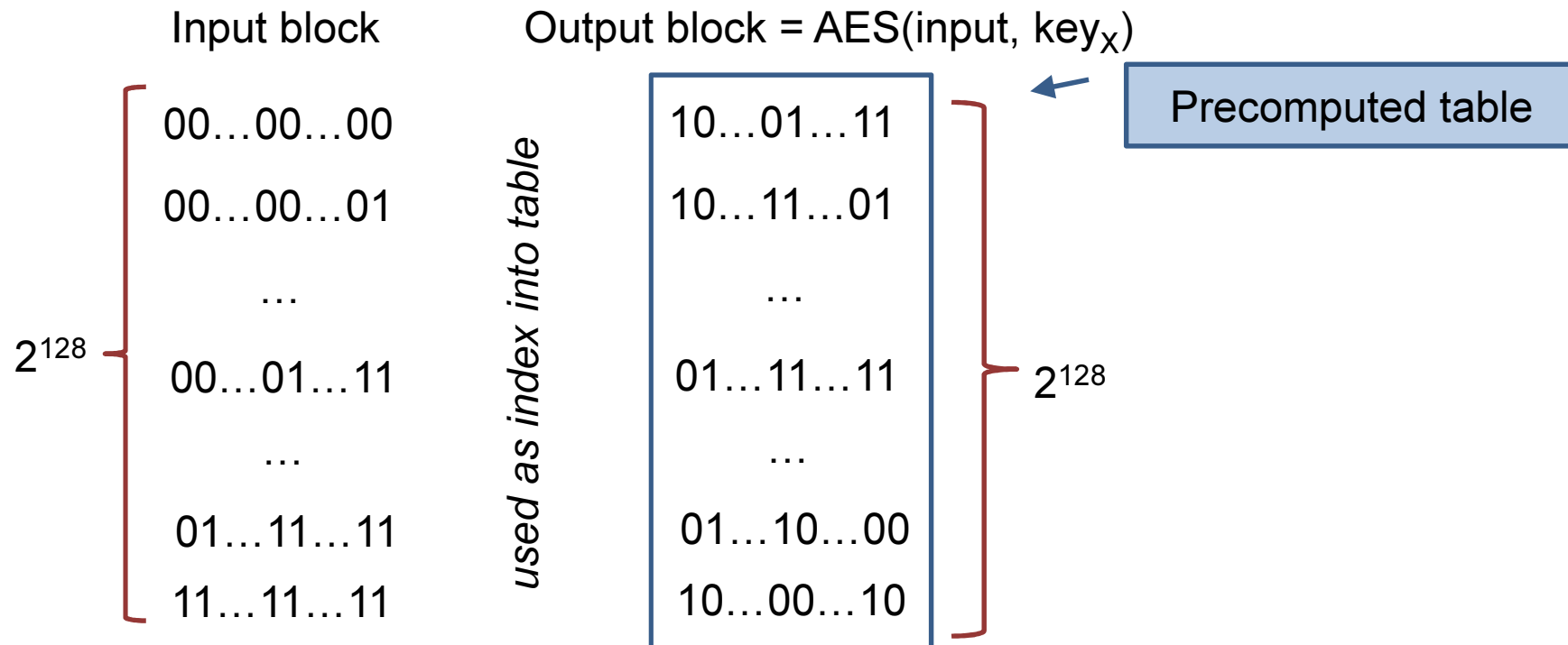
WHITEBOX RESISTANT CRYPTO

White-box attack resistant cryptography

- Problem limited from every cipher to symmetric cryptography cipher only
 - protects used cryptographic key (and data)
- Special implementation fully compatible with standard AES/DES... 2002 (Chow et al.)
 - series of lookups into pre-computed tables
- Implementation of AES which takes only data
 - key is already embedded inside
 - hard for an attacker to extract embedded key



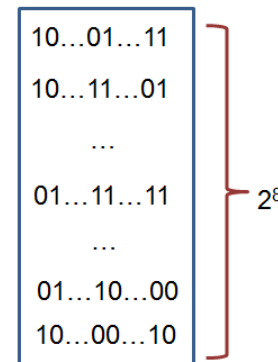
Impractical solution



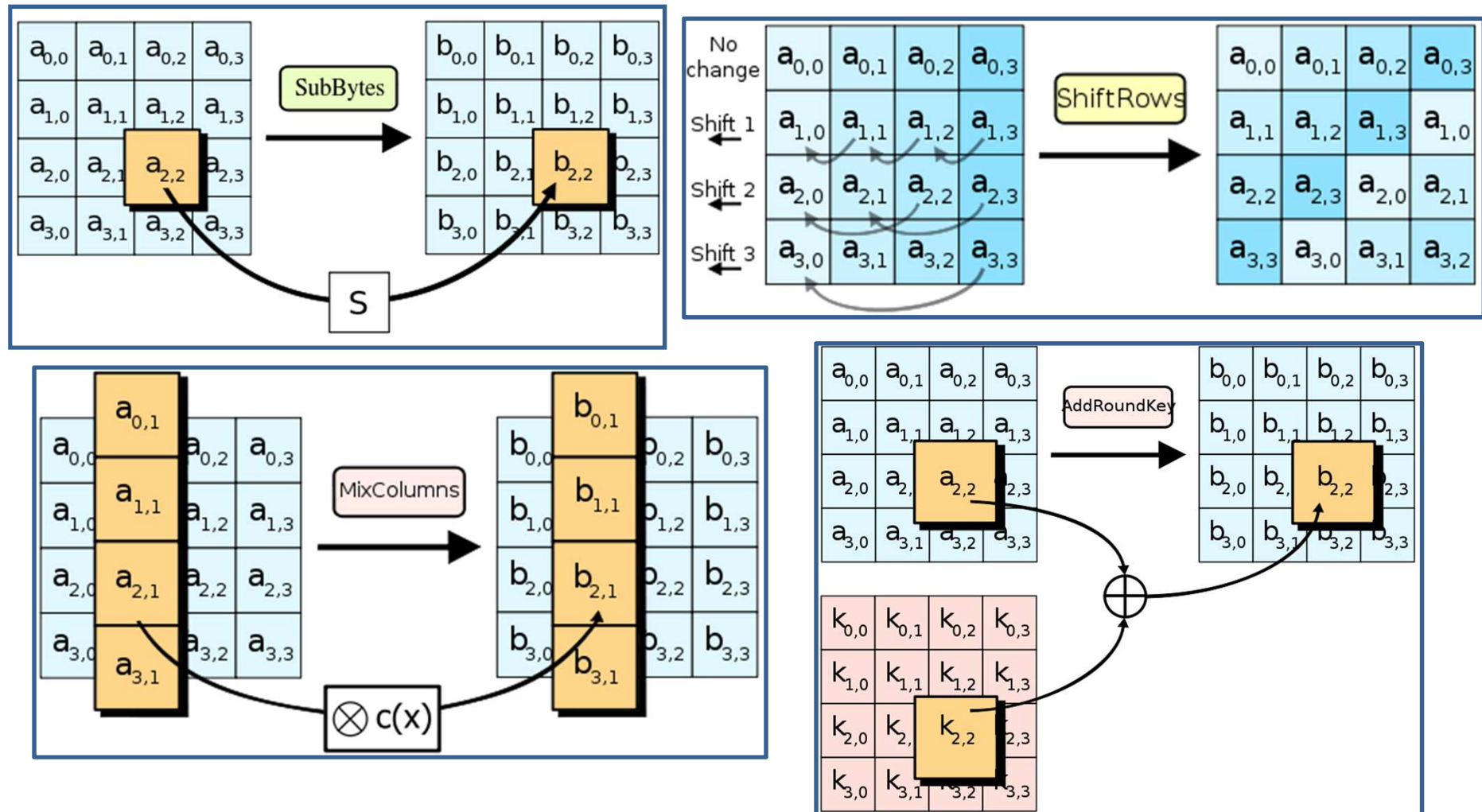
- Secure, but $2^{128} \times 16\text{B}$ memory storage

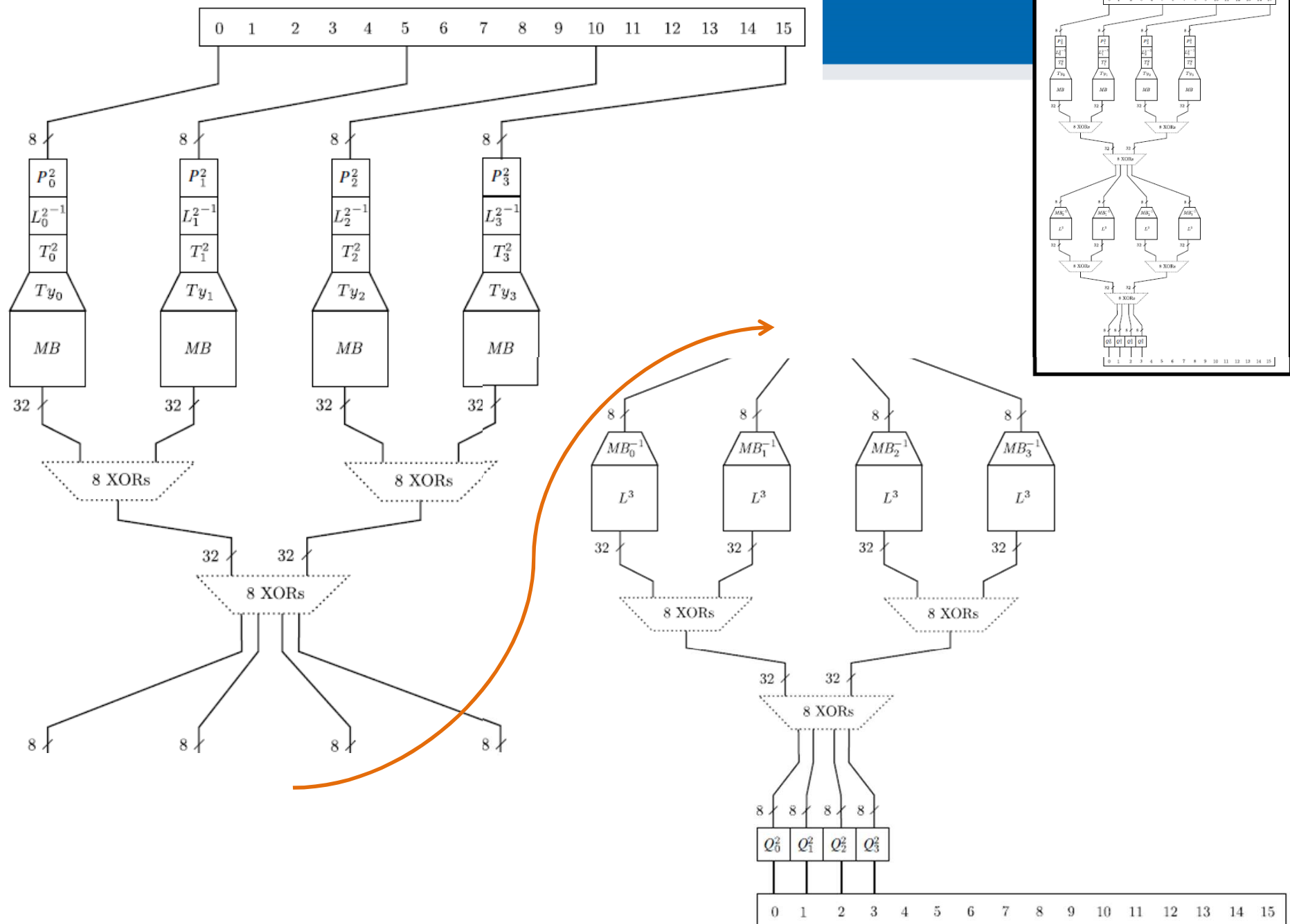
WBACR AES – some techniques

- Pre-compute table for all possible inputs
 - practical for one 16bits or two 8bits arguments table with up to 2^{16} rows (~64KB)
 - **AddRoundKey**: $\text{data} \oplus \text{key}$
 - 8bit argument data, key fixed
- Pack several operations together
 - **AddRoundKey+SubBytes**: $T[i] = S[i \oplus \text{key}] ;$
- Protect intermediate values by random bijections
 - removed automatically by next lookup
 - $X = F^{-1}(F(X))$
 - $T[i] = S[F^{-1}(i) \oplus \text{key}] ;$



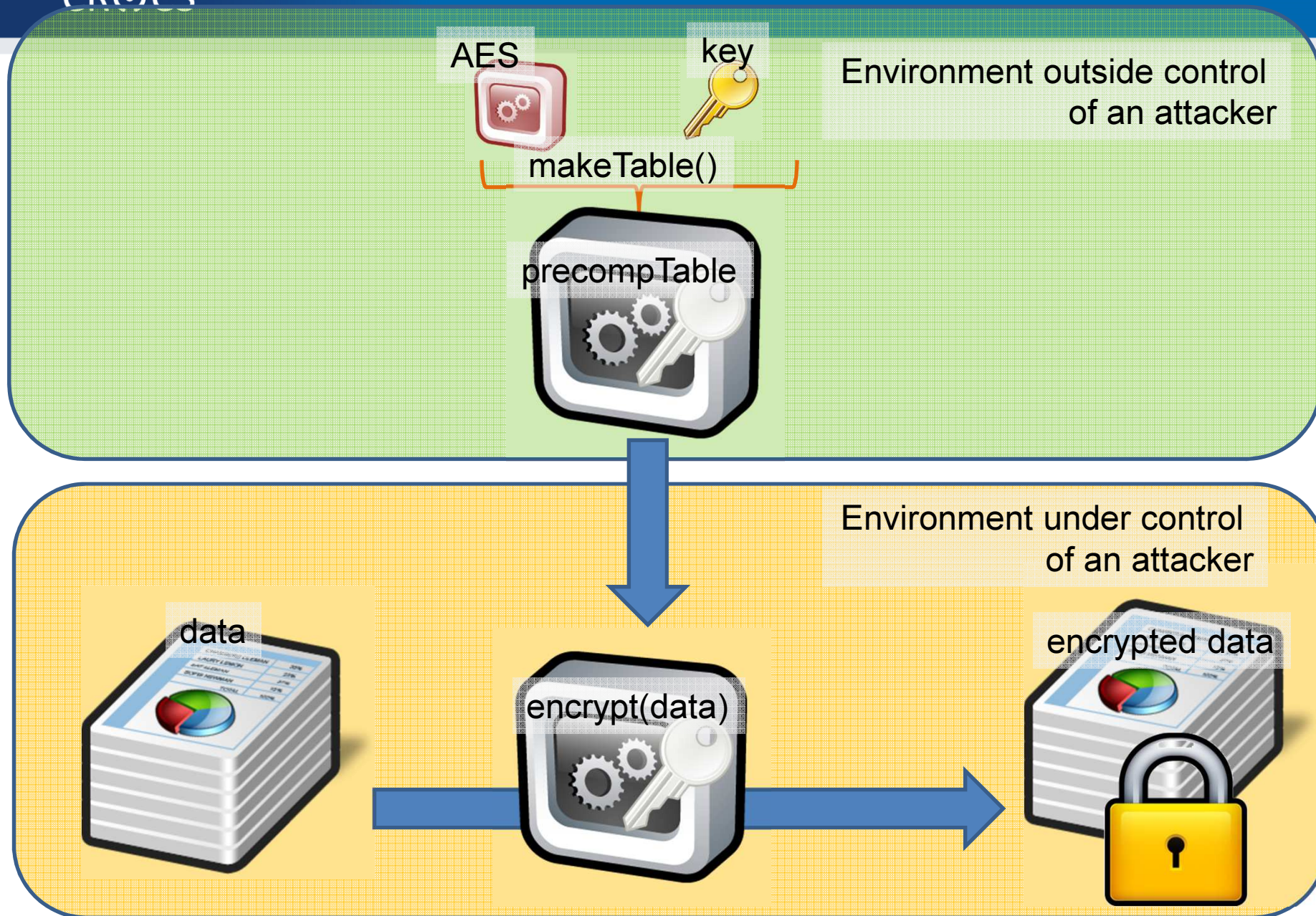
AES – short remainder (used ops)





Whitebox cryptography lifecycle

- [Secure environment]
 1. Generate required key (random, database...)
 2. Generate WAES tables (in secure environment)
- [Potential insecure environment]
 3. Compile WAES tables into target application
- [Insecure environment (User PC)]
 4. Run application and use WAES as usual (with fixed key)



Resulting implementation

- More difficult to detect that crypto was used
 - no fixed constants in the code
 - precomputed tables change with every generation
 - even two tables for same key are different
 - (but can still be found)
- Resistant even when precomputed tables are found
 - when debugged, only table lookups are seen
 - key value is never manipulated in plaintext
 - transformation techniques should provide protection to key embedded inside tables

WBACR AES - pros

- Practically usable
 - implementation size ~800KB (tables)
 - speed ~MBs/sec (~6.5MB/s vs. 220MB/s)
- Hard to extract embedded key
 - Complexity semi-formally guaranteed
 - (if the scheme is secure)
- One can simulate asymmetric cryptography!
 - implementation contains only encryption part of AES
 - until attacker extracts key, decryption is not possible

WBACR AES - cons

- Implementation can be used as oracle (black box)
 - attacker can supply inputs and obtain outputs
 - even if she cannot extract the key
 - (can be partially solved by I/O encodings)
- Problem of secure input/output
 - protected is only AES, not code around
- Key is fixed and cannot be easily changed
- Successful cryptanalysis for several schemes
 - several former schemes broken
 - new techniques proposed

Can whitebox transform replace secure hardware (e.g., smart card)?

- Only to limited extent
- Limitation of arguments size
- Operation atomicity
 - one cannot execute only half of card's operations
- No secure memory storage
 - no secure update of state (counter)
- Both can be used as black-box
 - smart card can use PIN to limit usage
- But still some reasonable usages remain

List of proposals and attacks

- (2002) First WB AES implementation by Chow et. al. [Chow02]
 - IO bijections, linear mixing bijections, external coding
 - broken by BGE cryptanalysis [Bill04]
 - algebraic attack, recovering symmetric key by modelling round function by system of algebraic equations
- (2006) White Box Cryptography: A New Attempt [Bri06]
 - attempt to randomize whitebox primitives, perturbation & random equations added, S-boxes are enc. keys. 4 AES ciphers, major voting for result
 - broken by Mulder et. al. [Mul10]
 - removes perturbations and random equations, attacking on final round removing perturbations, structural decomposition. 2^{17} steps
- (2009) A Secure Implementation of White-box AES [Xia09]
 - broken by Mulder et. al. [Mul12]
 - linear equivalence algorithm used (backward AES-128 compatibility => linear protection has to be inverted in next round), 2^{32} steps
- (2011) Protecting white-box AES with dual ciphers [Kar11]
 - broken by our work [Kli13]
 - protection shown to be ineffective

More resources

- Overviews, links
 - <http://whiteboxcrypto.com/research.php>
 - <https://minotaur.fi.muni.cz:8443/~xsvenda/docuwiki/doku.php?id=public:mobilecrypto>
- Crackme challenges
 - <http://www.phrack.org/issues.html?issue=68&id=8>
- Whitebox crypto in DRM
 - http://whiteboxcrypto.com/files/2012_MISC_DRM.pdf

Whitebox transform IS used in the wild

- Proprietary DRM systems
 - details are usually not published
 - AES-based functions, keyed hash functions, RSA, ECC...
 - interconnection with surrounding code
- Chow et al. (2002) proposal made at Cloakware
 - firmware protection solution
- Apple's FairPlay & Brahms attack
 - http://whiteboxcrypto.com/files/2012_MISC_DRM.pdf
- TrojanSpy:Win32/WhiteBox? ☺
- ...

Available practical implementations

DEMO

Demo – WAES

- WAES tables generator
 - configuration options
 - *.h files with pre-computed tables
- WAES cipher implementation
 - compile-in tables
 - tables as memory blob

```
# Encryption and decryption key values. Used during "/g:" option .
[KEY_VALUE]
# Key value which will be used for encryption part of WBACR AES tables.
# CAN be different from decryptKey
encryptKey=8a b1 21 d3 13 d1 5e 31 29 84 4c 66 50 14 6e 95
# Key value which will be used for decryption part of WBACR AES tables.
decryptKey=00 01 02 03 05 06 07 08 0A 0B 0C 0D 0F 10 11 12
# Additional entropy used for WBACR DES tables pre-computation
entropy=4d 28 a7 cd f9 c6 64 bc 94 c4 0e 77 79 7a 41 dc d8 19 9a 4b 0e 1c
```



```
#ifndef AESINVFIRSTTABLE_AUTH_H
#define AESINVFIRSTTABLE_AUTH_H

BYTE          invFirstRoundTable_auth[4][4][256] = {
{
{0x23, 0xee, 0x4c, 0x94, 0x4e, 0x32, 0x95, 0x3d, 0xa6, 0x
0xff, 0x34, 0x8e, 0x39, 0xcb, 0x82, 0x43, 0x87, 0x9b, 0xe
0xf, 0xc1, 0xaf, 0x1e, 0x6b, 0x8f, 0xbd, 0x2, 0xca, 0x8a,
0xc7, 0xb1, 0x12, 0xa8, 0x5f, 0x33, 0x10, 0x31, 0x88, 0xe
0xd6, 0xe1, 0x69, 0x4, 0x7d, 0x7e, 0x14, 0x26, 0xba, 0xc,
0x35, 0xe2, 0xf9, 0x74, 0x6e, 0x22, 0x37, 0x85, 0xe7, 0xc
0x24, 0x76, 0x5b, 0xa1, 0x25, 0x66, 0xa2, 0xb2, 0x28, 0xc
0x79, 0x9a, 0xdb, 0x3e, 0xf4, 0x4b, 0xc0, 0x20, 0xc6, 0x5
0xf5, 0xc8, 0xeb, 0x3b, 0x61, 0x4d, 0xbb, 0xb0, 0xae, 0x5
0xb9, 0x5e, 0x15, 0x48, 0x84, 0x50, 0x46, 0xda, 0xfd, 0xc
```

WAES performance

- Intel Core i5 M560@2.67GHz

Test	Result	Additional info.	OpenSSL result
generate WB AES	8.48 s avg.	100 samples	
throughput, 1 MB random	867.8 KB/s	1.18 s	57283 KB/s
throughput, 10 MB random	1022.977 KB/s	10.01 s	54179 KB/s
throughput, 100 MB random	1028.319 KB/s	99.58 s	74744 KB/s
throughput, 1024 MB random	1124.792 KB/s	932.24 s	63723 KB/s
throughput, 1 MB null	975 KB/s	1.05 s	93091 KB/s
throughput, 10 MB null	969.970 KB/s	10.56 s	68821 KB/s
throughput, 100 MB null	1058.507 KB/s	96.74 s	56356 KB/s
throughput, 1024 MB null	1050.593 KB/s	998.08 s	57283 KB/s

Table 4.2: Results of the benchmark for whitebox AES generator

BGE attack in progress

```
recoverQj; q = 0x88; gamma=0x01;
recoverQj self-test; r=5; col=3; (y0, y3); P[0].deltaInv=0x03; alfa_{3,0}=0x03
recoverQj self-test; r=5; col=3; (y0, y3); P[1].deltaInv=0x01; alfa_{3,1}=0x01
recoverQj self-test; r=5; col=3; (y0, y3); P[2].deltaInv=0x01; alfa_{3,2}=0x01
recoverQj self-test; r=5; col=3; (y0, y3); P[3].deltaInv=0x02; alfa_{3,3}=0x02
recoverQj; q = 0x3c; gamma=0x01;
```

Going to reconstruct encryption key from extracted round keys...

* Round keys extracted from the process, r=3

```
0x3d 0x47 0x1e 0x6d 0x80 0x16 0x23 0x7a 0x47 0xfe 0x7e 0x88 0x7d 0x3e 0x44 0x3b
```

* Round keys extracted from the process, r=4

```
0xef 0xa8 0xb6 0xdb 0x44 0x52 0x71 0x0b 0xa5 0x5b 0x25 0xad 0x41 0x7f 0x3b 0x00
```

* Round keys extracted from the process, r=5

```
0xd4 0x7c 0xca 0x11 0xd1 0x83 0xf2 0xf9 0xc6 0x9d 0xb8 0x15 0xf8 0x87 0xbc 0xbc
```

Recovering cipher key from round keys...

We have correct Rcon! rconIdx=3

RC=2; previousKey:

```
0xf2 0x7a 0x59 0x73
0xc2 0x96 0x35 0x59
0x95 0xb9 0x80 0xf6
0xf2 0x43 0x7a 0x7f
```

RC=1; previousKey:

```
0xa0 0x88 0x23 0x2a
0xfa 0x54 0xa3 0x6c
0xfe 0x2c 0x39 0x76
0x17 0xb1 0x39 0x05
```

RC=0; previousKey:

```
0x2b 0x28 0xab 0x09
0x7e 0xae 0xf7 0xcf
0x15 0xd2 0x15 0x4f
0x16 0xa6 0x88 0x3c
```

Final result:

```
0x2b 0x7e 0x15 0x16 0x28 0xae 0xd2 0xa6 0xab 0xf7 0x15 0x88 0x09 0xcf 0x4f 0x3c
```

Benchmark finished! Total time = 3a s; on average = 58 s; clocktime=57,66 s;

What's in our pipeline?

FUTURE WORK

Webpage with implemented proposals

- Obvious next step 😊
- Relevant academic papers didn't come with implementation 😞
 - true both for proposals and attacks
- Our work provided 2 implementations & 2 attacks
 - we will do remaining soon
- Relevant links
- CrackMe challenges
- <http://www.fi.muni.cz/~xsvenda/whiteboxcrypto/>

Modifications to W-AES

- Break backward AES compatibility → new cipher
 - but same scheme, strong primitives, key dependency
- 1. Hash-chain generated round keys
 - noninvertible
- 2. Key-dependent confusion / S-boxes
 - high variability (13 bytes dependence)
- 3. Key-dependent diffusion
 - $32 \times 32 \rightarrow 128 \times 128$ matrix
- 4. Incorporating of algebraic incompatible operations
 - like in IDEA cipher

Automatic white-box code transformation

- Parse existing source code
- Identify “transformable” operations
 - suitable size of operands
 - no side effects
 - ...
- Transform operations into white-box representation
 - or move to smart card
- Update existing code accordingly

```

for (i = start; i < end; i += 2) {
    int16_t cbits = 0;
    uint16_t xbits = 0;
    unsigned int xlen = h->xlen;
    unsigned int ext = 0;
    unsigned int x1 = gi->l3_enc[i];
    unsigned int x2 = gi->l3_enc[i + 1];

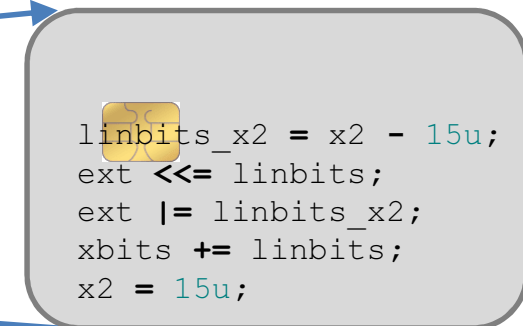
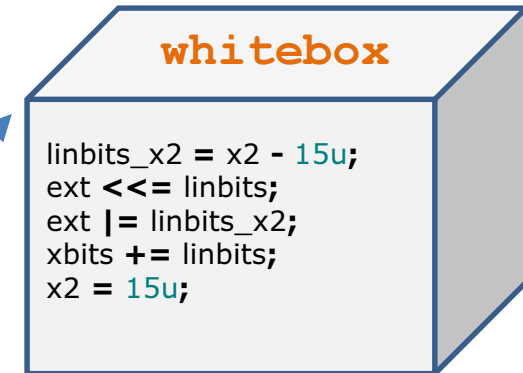
    assert(gi->l3_enc[i] >= 0);
    assert(gi->l3_enc[i+1] >= 0);

    if (x1 != 0u) {
        if (gi->xr[i] < 0.0f)
            ext++;
        cbits--;
    }

    if (tableindex > 15u) {
        /* use ESC-words */
        if (x1 >= 15u) {
            uint16_t const linbits_x1 = x1 - 15u;
            assert(linbits_x1 <= h->linmax);
            ext |= linbits_x1 << 1u;
            xbits = linbits;
            x1 = 15u;
        }

        if (x2 >= 15u) {
            uint16_t const linbits_x2 = x2 - 15u;
            assert(linbits_x2 <= h->linmax);
            ext <<= linbits;
            ext |= linbits_x2;
            xbits += linbits;
            x2 = 15u;
        }
        xlen = 16;
    }
}

```



Summary

- Computation with encrypted data & function
 - strong whitebox attacker model
- Whitebox cryptography tries to be better than classical obfuscation alone
 - mathematical-level proofs for cryptographic primitives
- Implementation of selected schemes (almost 😊) released
 - published attacks as well

Questions ? 

Thank you for your attention!

Questions ?



